Optimizing the Idle Task and Other MMU Tricks

Cort Dougan Paul Mackerras Victor Yodaiken

Department of Computer Science
New Mexico Institute of Technology
Socorro, NM 87801
{cort,yodaiken}@cs.nmt.edu *
paulus@cs.anu.edu.au

Abstract

In highly cached and pipelined machines, operating system performance, and aggregate user/system performance, is enormously sensitive to small changes in cache and TLB hit rates. We have implemented a variety of changes in the memory management of a native port of the Linux operating system to the PowerPC architecture in an effort to improve performance. Our results show that careful design to minimize the OS caching footprint, to shorten critical code paths in page fault handling, and to otherwise take full advantage of the memory management hardware can have dramatic effects on performance. Our results also show that the operating system can intelligently manage MMU resources as well or better than hardware can and suggest that complex hardware MMU assistance may not be the most appropriate use of scarce chip area. Comparative benchmarks show that our optimizations result in kernel performance that is significantly better than other monolithic kernels for the same architecture and highlight the distance that micro-kernel designs will have to travel to approach the performance of a reasonably efficient monolithic kernel.

1 Motivation

In the development of the PowerPC port of the Linux operating system, we have carried out a series of optimizations that has improved application wall-clock performance by anywhere from 10% to several orders of magnitude. According to the LmBench [5] benchmark, Linux/PPC is now twice as fast as IBM's AIX/PPC operating system and between 10 and 120 times faster than Apple's Mach based MkLinux/PPC and Rhapsody/PPC operating systems. We have achieved this level of performance by extensive use of quantitative measures and detailed analysis of low level system performance — particularly regarding memory management. While many of our optimizations have been machine specific, most of our results can be easily transported to other modern architectures and, we believe, are interesting both to operating system developers and hardware designers.

Our optimization effort was constrained by a requirement that we retain compatibility with the main Linux kernel development effort. Thus, we did not consider optimizations that would have required major changes to the (in theory) machine independent Linux core. Given this constraint, memory management was the obvious starting point for our investigations, as the critical role of memory and cache behavior for modern processor designs is well known. For commodity PC systems, the slow main memory systems and buses intensify this effect. What we found was that system performance was enormously sensitive to apparently small changes in the organization of page tables, in how we control the translation look aside buffer (TLB) and apparently innocuous OS operations that weakened locality of memory references. We also found that having a repeatable set of benchmarks was an invaluable aid in overcoming intuitions about the critical performance issues.

Our main benchmarking tools were the LmBench program developed by Larry McVoy and the standard Linux benchmark: timing and instrumenting a complete recompile of the kernel. These benchmarks tested aspects of system behavior that experience has shown to be broadly indicative for a wide range of applications. That is, performance improvements on the benchmarks seem to correlate to wall-clock performance improvements in application code. Our benchmarks do, however, ignore some important system behaviors and we discuss this problem below.

The experiments we cover here are the following:

- Reducing the frequency of TLB and secondary page map buffer misses.
  - Reducing the OS TLB footprint.
  - Increasing efficiency of hashed page tables.
- Reducing the cost of TLB misses.
  - Fast TLB reload code.
  - Removing hash tables — the ultimate optimization.
- Reducing TLB and page map flush costs.

[1]This work was partially funded by Sandia National Labs Grant 29P-9-TG100

  - Lazy purging of invalid entries.
  - Trading off purge costs against increased misses.
- Optimizing the idle task!

2 Related Work

There has been surprisingly little published experimental work on OS memory management design. Two works that had a great deal of influence on our work were a series of papers by Bershad [7] advocating the use of *"superpages"* to reduce TLB contention and a paper by Liedtke [3] on performance issues in the development of the L4 microkernel. Our initial belief was that TLB contention would be a critical area for optimization and that the monolithic nature of the Linux kernel would allow us to gain much more than was possible for L4.

It is the current trend in chip design to keep TLB size small, especially compared to the size of working sets in modern applications. There are many proposed solutions to the problem of limited TLB reach, caused by the disparity between application access patterns and TLB size, but most of them require the addition and use of special purpose hardware. Even the simpler proposed solutions require that the hardware implement superpages.

Swanson proposes a mechanism in [10] that adds another level of indirection to page translation to create noncontiguous and unaligned superpages. This scheme makes superpage use far more convenient since the memory management system does not have to be designed around it (finding large, continuous, aligned areas of unused memory is not easy).

Since existing hardware designs are set and the trend in emerging designs is towards relatively small TLBs we cannot rely on superpage type solutions or larger TLBs. More effective ways of using the TLB and greater understanding of TLB access patterns must be found. Greater understanding of access patterns and better ways of using TLB would only augment systems with superpages or larger TLBs.

3 A Quick Introduction to the PPC Memory Management System

Our OS study covers the 32-bit PowerPC 603 [6] and 604 processors. Both machines provide a modified inverted page table memory management architecture. The standard translation between logical and physical addresses takes place as follows:

Program memory references are 32-bit *logical* addresses. The 4 high order bits of the logical address index a set of segment registers, each of which contains a 24-bit *"virtual segment identifier"*(VSID). The logical address is concatenated with the VSID to produce a *virtual* address. There is a translation look-aside buffer of cached virtual $\rightarrow$ physical translations and *hashed page tables* indexed by a (hashed) virtual address. The tables are organized into "buckets", each consisting of eight page table entries (PTEs). Each

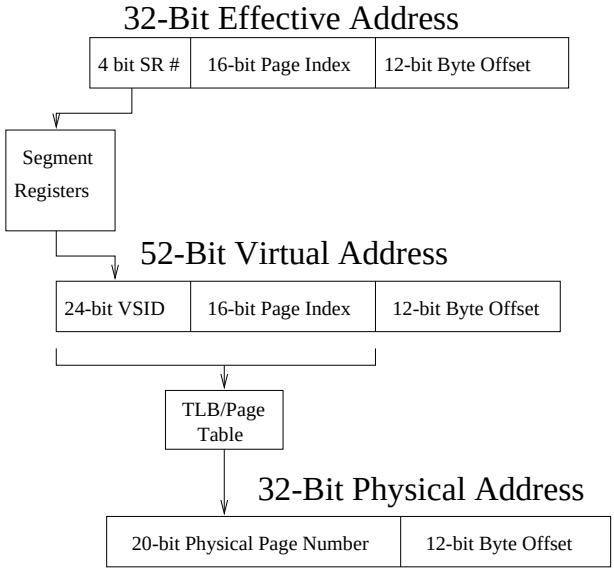

Figure 1: PowerPC hash table translation

PTE contains a 20-bit physical page address, a 24-bit virtual segment identifier (VSID) and permission and other housekeeping information. Once a TLB miss occurs, a hash function is computed on the virtual address to obtain the index of a bucket. If no matching entry is found in this bucket, a secondary hash function is computed to find the index of an overflow bucket. If no entry is found in either bucket, the OS must determine further action. On the 604, a TLB miss causes the hardware to compute the hash function and search the hash table. If no match is found, a page fault interrupt is generated and a software handler is started. On the 603, there are registers to assist hashing even though the hardware does not require software to store PTEs in a hash table. Since a TLB miss is handled in hardware, the 604 has a hash-table miss interrupt rather than a TLB miss interrupt.

The PowerPC also offers an alternative translation from logical to physical that bypasses the TLB/hash-table paging mechanism. When a logical address is referenced, the processor begins the page lookup and, in parallel, begins an operation called *block address translation*(BAT). Block address translation depends on eight BAT registers: four data and four instruction. The BAT registers associate virtual blocks of 128K or more with physical segments. If a translation via the BAT registers succeeds, the page table translation is abandoned.

4 Performance Measurement

Benchmarks and tests of performance were made on a number of PowerPC processors and machine types (PReP and PowerMac) to reduce the amount any specific machine would affect measurements. We used 32M of RAM in each machine tested. This way, the ratio of RAM size to PTEs in the hash table to TLB entries remained the same. Each of the test results comes from more than 10 of the benchmark

runs averaged. We ignore benchmark differences that were sporadic even though we believe this understates the extent of our optimizations.

Tests were made using LmBench [5]. We also used the informal Linux benchmark of compiling the kernel, which is a traditional measure of Linux performance. The mix of process creation, file I/O, and computation in the kernel compile is a good guess at a typical user load in a system used for program development.

Performance comparisons were made against various versions of the kernel. In our evaluations we compare the kernel against the original version without the optimizations discussed in this paper. This highlights each optimizations alone without the others. This lets us look more closely at how each change affects the kernel by itself before comparing all optimizations in aggregate. This turned out to be very useful as many optimizations did not interact as we expected them to and the end effect was not the sum off all the optimizations. Some optimizations even cancelled the effect of previous ones. So, measurements are relative to the original (unoptimized) kernel versus only the specific optimization being discussed for comparison unless otherwise noted.

Finally, we gathered low-level statistics with the PPC 604 hardware monitor. Using this monitor we were able to characterize the system's behavior in great detail by counting every TLB and cache miss, whether data or instruction. Software counters on the 603 were used to serve in much the same fashion as hardware performance monitoring hardware on the 604, but with a less fine-grained scope.

We make many references to the 603 software versus the 604 hardware TLB reload mechanism. In this context, when we refer to the 604 we mean the 604 style of TLB reloads (in hardware) which includes the 750 and 601.

5 Reducing the Frequency of TLB Misses

The 603 generates software handled interrupts on a TLB miss. It takes 32 cycles simply to invoke and return from the handler — ignoring the costs of actually reloading the TLB. The 604 PPC executes a hardware hash table search on a TLB miss. If the PTE is in the hash table (the PowerPC page-table), the cost of the hardware reload can take up to 120 instruction cycles (measured experimentally) and 16 memory accesses. If the hash table does not contain the PTE, the 604 generates a software handled interrupt that adds at least 91 more cycles to just invoke the handler. With interrupt overhead this high, TLB reloads will be expensive no matter how much we optimize the TLB reload code itself. With this motivation, we thought it worthwhile to reduce the frequency of TLB misses as much as possible.

5.1 Reducing the OS TLB footprint

Bershad [7] and others have argued on theoretical grounds that "superpages" and other mechanisms for reducing the OS TLB footprint can greatly improve performance. Indeed, we found that 33% of the TLB entries under Linux/PPC were for kernel text, data and I/O pages. The PowerPC 603 TLB has 128 entries and the 604 has 256 entries, so allocating a third of the entries to the OS should have a significant effect on performance. While some processor architectures (MIPS [2]) directly support superpage schemes, the PPC does not. There are too few BAT registers and their granularity is too coarse for a straightforward superpage implementation. The BATs can, however, still be used to reduce the number of entries taken up in the page-table and, therefore, reduce the number of TLB misses. Since user processes are often ephemeral and large block sizes for each user process would waste physical memory, we decided to begin by using the BAT mapping only for kernel address space.

Linux, like many other UNIX variants, divides each user processes virtual address space into two fixed regions: one for user code and data and one for the kernel. On a 32 bit machine, the Linux kernel usually resides at virtual address `0xc0000000` and virtual addresses from `0xc0000000` to `0xffffffff` are reserved for kernel text/data and I/O space. We began by mapping all of kernel memory with PTEs. We quickly decided we could reduce the overhead of the OS by mapping the kernel text and data with the BATs. The kernel mappings for these addresses do not change and the kernel code and static data occupy a single contiguous chunk of physical memory. So, a single BAT entry maps this entire address space. Note that one side effect of mapping kernel space via the BATs is that the hash tables and backing page tables do not take any TLB space. Mapping the hash table and page-tables is given to us for free so we don't have to worry about recursively faulting on a TLB miss.

Using the BAT registers to map kernel space on the kernel compile we measure a 10% reduction in TLB misses (from 219 million to 197 million TLB misses on average) and a 20% reduction in hash table misses (from an average 1 million hash table misses to 813 thousand hash table misses) during our benchmarks. The percentage of TLB slots occupied by the kernel dropped to near zero — the high water mark we have measured for kernel PTE use is four entries. The kernel compile benchmark showed a 20% reduction in wall-clock time - from 10 to 8 minutes. Using the BAT registers to map the I/O space did not improve these measures significantly. The applications we examined rarely accessed a large number of I/O addresses in a short time so it is rare that the TLB entries are mapping I/O areas since they are quickly displaced by other mappings. We have considered having the kernel dedicate a BAT mapping to the frame buffer itself so programs such as X do not compete constantly with other applications or the kernel for TLB space. In fact, the entire mechanism could be done per-process with a call to `ioremap()` and giving each process its own data BAT entry that could be switched during a context switch.

Much to our chagrin, nearly all the measured performance improvements we found from using the BAT registers evaporated when TLB miss handling was optimized. That is, the TLB misses caused by kernel - user contention

are few enough so that optimizing reloads makes the cost of handling these reloads minimal — for the benchmarks we tried. In light of Talluri [11], however, it's quite possible that our benchmarks do not represent applications that really stress TLB capacity. More aggressive use of the TLB, such as several applications using many TLB entries running concurrently would possibly show an even greater performance gain. Not coincidentally, this optimizes for the situation of several processes running in separate memory contexts (not threads) which is the typical load on a multiuser system.

5.2 Increasing the Efficiency of Hashed Page Tables

The core of Linux memory management is based on the $x86$ two-level page tables. We could change the organization of the PTEs in these tables to match the requirements of the PPC architecture (a hash table instead of a two-level page table), but we were committed to using these page tables as the initial source of PTE's due to the design of Linux. Note that any PPC OS must have a data structure to serve this function, because the PTEs that do not fit in either the primary or overflow bucket must be stored somewhere. It is possible, but impractical, to resize the hash table when both buckets overflow. Various techniques for handling overflow are discussed in [8] and [12]. A reasonable PPC OS must minimize the number of overflows in the hash table so the cost of handling overflows was not a serious concern for us. Instead, we focused on reducing the contention in the hash table to increase the efficiency of the hash table which reduces the number of overflows. Our original strategy for both the 603 and 604 processors was to use the hash table as a second level TLB cache and, thus, it became important to reduce hash table "hot spots" and overflow.

The obvious strategy is to derive VSIDs from the process identifier so that each process has a distinct virtual address space. Keep in mind that the hardware will treat each set of VSIDs as a separate address space. Multiplying the process id by a small non-power-of-two constant proved to be necessary in order to scatter PTEs within the hash table. Note that the logical address spaces of processes tend to be similar so the hash functions rely on the VSIDs to provide variation. We tuned the VSID generation algorithm by making Linux keep a hash table miss histogram and adjusting the constant until hot-spots disappeared. We began with 37% use of the hash table and were able to bring that up to an average of 57% with the hash table containing both user and kernel PTE's. After removing the kernel PTE's from the hash table we were eventually able to achieve 75% use of the hash table with find tuning of the constant.

6 Reducing the Cost of TLB and Hash Table Misses

6.1 Fast Reload Code

On an interrupt, the PowerPC turns off memory management and invokes a handler using physical addresses. Originally, we turned the MMU on, saved state and jumped to fault handlers written in C to search the hash table for the appropriate PTE. To speed the TLB reload we rewrote these handlers in assembly and hand optimized the TLB and hash table miss exception code for both 603 and 604 processors. The new handlers ran with the memory management hardware off and we tried to make sure that the reload code path was as short as possible.

Careful coding of miss handlers proved to be worth the effort. On an interrupt, the PPC turns off memory management and swaps 4 general purpose registers with 4 interrupt handling registers on a TLB miss. We rewrote the TLB miss code to use only these registers in the common case. Following the example of the Linux/SPARC developers, we also hand scheduled the code to minimize pipeline hiccups. The Linux PTE tree is sufficiently simple that searching for a PTE in the tree can be done conveniently with the MMU disabled, in assembly code, and taking three loads in the worst case. If the PTE cannot be found at all or if the page is not in memory, we turn on memory management, save additional context and jump to C code.

These changes produced a 33% reduction in context switch time and reduced communication latencies by 15% as measured with LmBench. User code showed an improvement of 15% in general when measured by wall-clock time.

6.2 Improving Hash Tables Away

The 603 databook recommends using hardware hashing assists to emulate the 604 behavior on the 603. Following this recommendation, the early Linux/PPC TLB miss handler code searched the hash table for a matching PTE. If no match was found, software would emulate a hash table miss interrupt and the code would execute as if it were on a 604 that had done a search in hardware. Our conjecture was that this approach simply added another level of indirection and would cause cache misses as the software stumbled about the hash table.

The optimization we tried was to eliminate any use of the hash table and to have the TLB miss handler go directly to the Linux PTE tree. By following this strategy we make a 180MHz 603 keep pace with a 185MHz 604 despite the two times larger L1 cache and TLB in the 604. In fact, on some LmBench points, the 180MHz 603 kept pace with a 200MHz 604 on a machine with significantly faster main memory and a better board design. Unfortunately, the 604 does not permit software to reload the TLB directly, which would allow us to make this optimization on the 604. The end result of these changes was a kernel compile time reduced by 5%.

processor	pstart	ctxsw	pipe lat.	pipe bw	file reread
603 180MHz (htab)	1.8s	4 μs	17 μs	69 MB/s	33 MB/s
603 180MHz (no htab)	1.7s	3 μs	19 μs	73 MB/s	36 MB/s
604 185MHz	1.6s	4 μs	21 μs	88 MB/s	39 MB/s
604 200MHz	1.6s	4 μs	20 μs	92 MB/s	41 MB/s

Table 1: LmBench summary for direct (bypassing hash table) TLB reloads

Using software TLB reloads which are available on many platforms, such as the Alpha [9], MIPS [2] and UltraSPARC, allows the operating system designer to consider many different page-table data structures (such as clustered page tables [11]). If the hardware doesn't constrain the choices many optimizations can be made depending on the type of system and typical load the system is under.

7 Reducing TLB and Hash Table Flush Costs

Because the 604 requires us to use the hash table, any TLB flush must be accompanied by a hash table flush. Linux flushes all or part of a process's entries in the TLB frequently, such as when mapping new addresses into a process, doing an `exec()` or `fork()` and when a dynamically linked Linux process is started, the process must remap its address space to incorporate shared libraries. In this context, a TLB flush is actually a TLB invalidate since we updated the page-table PTE dirty/modified bits when we loaded the PTE into the hash table. In the worst case, the search requires 16 memory references (2 hash table buckets, containing 8 PTE's each) for each PTE being flushed. It is not uncommon for ranges of 40 — 110 pages to be flushed in one shot.

The obvious strategy, and the first one we used, was for the OS to derive VSIDs from the process identifier (so that each process has a distinct virtual address space) and multiplying it by a scalar to scatter entries in the hash table. After doing this, we found that flushing the hash table was extremely expensive, we then came upon the idea of lazy TLB flushes. Our idea of how to do lazy TLB flushes was to keep a counter of memory-management contexts so we could provide unique numbers for use as VSID's instead of using the PID of a process. We reserved segments for the dynamically mapped parts of the kernel (static areas, data and text, are mapped by the BATs) and put a fixed VSID in these segments. When the kernel switched to a task its VSIDs could be loaded from the task structure into hardware registers by software.

When we needed to clear the TLB of all mappings associated with a particular task we only had to change the values (VSIDs) in the task structure and then update the hardware registers and increment the context counter. Even though there could be old PTEs from the previous context (previous VSIDs) in the hash table and TLB marked as valid PTEs (valid bit set in the PTE) their VSID's will not match any VSID's used by any process so incorrect matches won't be made. We could then keep a list of "zombie" VSID's (similar to "zombie" processes) that are marked as valid in the hash table but aren't actually used and clear them when hash table space became scarce.

What we finally settled on and implemented was different from what we had planned at first. Deciding when to really flush the old PTE's from the hash table and how to handle the "zombie list" was more complex than we wanted to deal with at first. Performance would also be inconsistent if we had to occasionally scan the hash table and invalidate "zombie" PTE's when we needed more space in the hash table. So, instead, we just allowed the hash table reload code to replace an entry when needed (not checking if it has a currently valid VSID or not). This gave a non-optimal replacement strategy in the hash table since we may replace real PTEs (have a currently active VSID) in the hash table even though there are PTEs that aren't being used (have the VSID of an old memory context).

We were later able to return to this idea of reducing the inefficiency of the hash table replacement algorithm (replacing unused PTE's from an abandoned memory management context marked as valid) by setting the idle task to reclaim zombie hash table entries by scanning the hash table when the cpu is idle and clearing the valid bit in zombie PTEs (physically invalidating those PTEs). This provided a nice balance of simplifying the actual low level assembly to reload the TLB and still maintaining a decent usage ratio of the hash table (zombie PTEs to in-use PTEs).

Without the code to reclaim zombie PTEs in the idle task, the ratio of hash table reloads to evicts (reloads that require a valid entry be replaced) was normally greater than 90%. Since the hot-spots were eliminated in the hash table, entries are scattered across all of the hash table and never invalidated. Invalidation in this case means the valid bit is never cleared even though the VSID may not match a current process. Very quickly the entire hash table fills up. Since the TLB reload code did not differentiate between the two types of invalid entries, it chose an arbitrary PTE to replace when reloading, replacing valid PTEs as well as zombie PTEs. With the reclaim code in the idle task, we saw a drastic decrease in the number of evicts. This is because the hash table reload code was usually able to find an empty TLB entry and was able to avoid replacing valid PTEs.

Our series of changes took hash table use up to 15% and finally down to around 5% since we effectively flush all the TLB entries of a process when doing large TLB flushes (by changing the VSID's of a process). Our optimization to re-

duce hot-spots in the hash table was not as significant since so few entries stayed in the hash table (about 600–700 out of 16384) at one time due to the flushing of ranges of PTEs. Even with this little of the hash table in use we measured 85% — 95% hit rates in the hash table on TLB misses. To increase the percentage of hash table use, we could have decreased the size of the hash table and free RAM for use by the system but in performing these benchmarks we decided to keep the hash table size fixed to make comparisons more meaningful. This choice makes the hash table look inefficient with some optimizations but the net gain in performance as measured by hit rate in the hash table and wall-clock time shows it is in fact an advantage.

Another advantage of the idle task invalidating PTEs was that TLB reload code was usually able to find an empty entry in the hash table during a reload. This reduced the number of evicts so the ratio of evicts to TLB reloads became 30% instead of the greater than 90% we were seeing before. This reduced number of evicts also left the hash table with more in-use PTEs so our usage of the hash table jumped to 1400–2200 from 600–700 entries, or to 15% from 5%. The hit rate in the hash table on a TLB miss also increased to as high as 98% from 85%.

Using lazy TLB flushes increased pipe throughput by 5 MB/s (from 71 MB/s) and reduced 8-process context switch times from 20μs to 17μs. However, the system continued to spend a great deal of time searching the hash table because for certain operations, the OS was attempting to clear a *range* of virtual addresses from the TLB and hash table. The OS must ensure that the new mappings are the only mappings for those virtual addresses in the TLB and hash table. The system call to change address space is `mmap` and LmBench showed `mmap()` latency to be more than 3 milliseconds. The kernel was clearing the range of addresses by searching the hash table for each PTE in turn. We fixed this problem by invalidating the whole memory management context of any process needing to invalidate more than a small set of pages. This effectively invalidates all of the TLB entries of this process. This was a cheap operation with the mechanism we used since it just involved a reset of the VSID whose amortized cost (later TLB reloads vs. cost of flushing specific PTEs) is much lower. Once the process received a new VSID and its old VSID was marked as zombie, all the process' PTEs in the TLB and hash table were automatically inactive. Of course, there is a performance penalty here as we invalidate some translations that could have remained valid, but using 20 pages as the cutoff point `mmap()` latency dropped to 41μ — an 80 times improvement. Pipe bandwidth increased noticeably and several latencies dropped as well. These changes come at no cost to the TLB hit rate since no more or fewer TLB misses occurred with the tunable parameter to flushing ranges of PTEs. This suggests that the TLB entries being invalidated along with target range of TLBs were not being used anyway - so there is no cost for losing them.

Table 2 shows the 603 doing software searches of the hash table and a 604 doing hardware searches with the effect of lazy TLB flushes. Note that the 603 hash table search is using software TLB miss handlers that emulate the 604 hardware search. This table shows the gain from avoiding expensive searches in the hash table when a simple resetting of the VSID's will do.

8 Cache Misuse on Page-Tables

Caching page-tables can be a misuse of the hardware. The typical case is that a program changes its working set, which adds a particular page. That page is referenced and its PTE is brought into the TLB from the page-tables and during that TLB reload the PTE is put in the data cache. On the 604 the PTE is also put into the hash table, which creates another cache entry. TLB reloads from page-tables are rare and caching those entries only pollutes the cache with addresses that won't be used for a long time.

Caching page tables makes sense only if we will make repeated references to adjacent entries, but this behavior does not occur nor is it common for page-table access patterns. For this to be common, we'd need to assume that either the page accesses (and PTEs used to complete those accesses) or cache accesses do not follow the principle of locality. The idea behind a TLB is that it's rare to change working sets and the same is true for a cache.

This is especially important under Linux/PPC since the hardware searched hash table (the PowerPC page-table) is actually used as a cache for the two level page table tree (similar to the SPARC and x86 tables). This makes it possible in the worst case for two separate tables to be searched in order to load one PTE into the TLB. This translates into 16 (hash table search and miss) + 2 (two level page table search) + 16 (finding a place to put the entry in the hash table) = 34 memory accesses in order to bring an entry into the hash table. If each one of these accesses is to a cached page there will be 18 new cache entries created (note that the first and second 16 hash table addresses accessed are the same so it is likely we will hit the cache on the second set of accesses).

By caching the page-tables we cause the TLB to pollute the cache with entries we're not likely to use soon. On the 604 it's possible to create two new cache entries that won't be used again due to accessing the hash table and the two-level page tables for the same PTE. This conflict is non-productive and causes many unnecessary cache misses. We have not yet performed experiments to quantify the number of cache misses caused by caching the page-tables but the results of our work so far suggests this has a dramatic impact on performance.

9 Idle Task Page Clearing

Clearing pages in the idle task is not a new concept but has not been popular due to its effect on the data cache. For the same reason we did not use the PowerPC instruction that clears entire cache lines at a time when we implemented $bzero()$ and similar functions. This problem illustrates the effect the operating system has on perfor-

processor	mmap lat.	ctxsw	pipe lat.	pipe bw	file reread
603 133MHz	3240 μs	6 μs	34 μs	52 MB/s	26 MB/s
603 133MHz (lazy)	41 μs	6 μs	28 μs	57 MB/s	32 MB/s
604 185MHz	2733 μs	4 μs	22 μs	90 MB/s	38 MB/s
604 185MHz (tune)	33 μs	4 μs	21 μs	94 MB/s	41 MB/s

Table 2: LmBench summary for tunable TLB range flushing

mance due to its use of the cache. We began by clearing pages in the idle task without turning off the cache for those pages. These pages were then added to a list which `get_free_page()` then used to return pages that had already been cleared without having to waste time clearing the pages when they were requested. The kernel compile with this "optimization" took nearly twice as long to complete due to cache misses. Measurements with LmBench showed performance decreases as well. The performance loss from clearing pages was verified with hardware counters to be due to more cache misses.

We repeated the experiment by uncaching the pages before clearing them and not adding them to the list of cleared pages. This allowed us to see how much of a penalty clearing the pages incurred without having the effect of using those pages to speed `get_free_page()`. There was no performance loss or gain. This makes sense since the data cache was not affected because the pages being cleared were uncached and even after being cleared they weren't used to speed up `get_free_page()`. The number of cache misses didn't change from the kernel without the page clearing modifications. [2]

When the cache was turned off for pages being cleared and they were used in `get_free_page()` the system became much faster. This kept the cache from having useless entries put into it when `get_free_page()` had to clear the page itself when the code requesting the page never read those values (it shouldn't read them anyway). This suggests that it might be worthwhile to turn off the data and instruction cache during the idle task to avoid polluting the cache with any accesses done in the idle task. There's no need to worry about the idle task executing quickly, we're only concerned with switching out of it quickly when another task needs to run so caching isn't necessary.

We must always ensure that the overhead of an optimization doesn't outweigh any potential improvement in performance [11] [4]. In this case we did not incur great overhead when clearing pages. In fact, all data structures used to keep track of the cleared pages are lock free and interrupts are left enabled so there is no possibility of keeping control of the processor any longer than if we had not been clearing pages. Even when calling `get_free_page()` the only overhead is a check to see if there are any pre-cleared pages available. Our measurements with page-clearing on but not adding the pages to the list still costs us that check in `get_free_page()` so any potential overhead would have shown up. This is important since the idle task runs quite often even on a system heavily loaded with users compiling, editing, reading mail so a lot of I/O happens that must be waited for.

[2] On a SMP machine we might see conflicts due to accesses causing cache operations on other processors or, more likely, all these writes to memory using a great deal of the bus while the other processor needs it

10 Future Work

There is still more potential in the optimizations we've already made. We've made these changes on a step-by-step basis so we could evaluate each change and study not only how it changed performance but why. There are still many changes we've seen that we haven't actually studied in detail that we believe to be worthy of more study. These include better handling of the cache in certain parts of the operating system and cache preloads.

10.1 Locking the Cache

Our experiments show that not using the cache on certain data in critical sections of the operating system (particularly the idle task) can improve performance. One area worthy of more research would be locking the cache entirely in the idle task. Since all of the accesses in the idle task (instruction and data) aren't time critical there's no need to evict cache entries just to speed up the idle task.

10.2 Cache Preloads

Waiting on the cache to load values from memory is a big performance loss. Modern processors rely on the fact that the pipeline will be kept full as much as possible but mishandling the cache can easily prevent this. One easy way to overcome this problem is to provide hints to the hardware about access patterns. On the PowerPC and other architectures there are sets of instructions to do preloads of data cache entries before the actual access is done. This is a simple way, without large changes, to improve cache behavior and reduce stalls. We feel that we can make significant gains with intelligent use of cache preloads in context switching and interrupt entry code.

11 Analysis

The results presented show that it is possible to achieve and exceed the speed of hardware TLB reloads with software handlers. This speedup depends very much on how well

OS	Null syscall	ctx switch	pipe lat.	pipe bw
Linux/PPC	2 μs	6 μs	28 μs	52 MB/s
Unoptimized Linux/PPC	18 μs	28 μs	78 μs	36 MB/s
Rhapsody 5.0	15 μs	64 μs	161 μs	9 MB/s
MkLinux	19 μs	64 μs	235 μs	15 MB/s
AIX	11 μs	24 μs	89 μs	21 MB/s

All tests except AIX performed on a 133MHz 604 PowerMac 9500, AIX number from a 133MHz 604 IBM 43P.

Table 3: LmBench summary for Linux/PPC and other Operating Systems

tuned the reload code is and what data structures are used to store the PTEs. On the 603 we find it is not necessary to mirror the same hash table that the hardware assumes on the 604. We can actually speed things up by eliminating the hash table entirely.

By reducing the conflict between user and kernel space for TLB entries we're able to improve TLB hit rates and speed the system up in general by about 20%. Our experiments bypassing the cache during critical parts of the operating system where creating new cache entries would actually reduce performance shows great promise. Already we're seeing fewer cache misses by avoiding creating cache entries for the idle task and expect to see even fewer with changes to the TLB reload code to uncache the page tables. We've been able to avoid expensive TLB flushes through several optimizations that bring `mmap()` latency down to a reasonable value - an 80 times improvement by avoiding unnecessary searches through the hash table.

Our hash table hit rate on a TLB miss is 80% – 98% which demonstrates that the hash table is well managed to speed page translations. We cannot realistically expect any improvement over this hit rate so our 604 implementation of the MMU is near optimal. When compared with our optimizations of the 603 MMU using software searches we're able to get better performance on the 603 in some benchmarks than the 604 even though the 604 has double the size TLB and cache.

All these changes suggest that cache and TLB management is important and the OS designer must look deeper into the interaction of the access patterns of TLB and cache. It isn't wise to assume that caching a page will necessarily improve performance.

Though it has been claimed [1] that micro-kernel designs can be made to perform as well as monolithic designs our data (Table 3) suggests that monolithic designs need not remain a stationary target.

The work we've mentioned has brought Linux/PPC to excellent standing among commercial and non-commercial offerings for operating systems. Table 3 shows our system compares very well with AIX on the PowerPC and is a dramatic improvement over the Mach-based Rhapsody and MkLinux from Apple.

The trend in processor design seems to be directed towards hardware control of the MMU. Designers of the hardware may see this as a benefit to the OS developer but it is, in fact, a hindrance. Software control of the MMU allows experimentation with different allocation and storage strategies but hardware control of the MMU is too inflexible. As a final note, the architects of the PowerPC series seem to have decided to increase hardware control of memory management. Our results indicate that they might better spend their transistors and expensive silicon real-estate elsewhere.

References

[1] *Personal Communication with Steve Jobs*. 1998.

[2] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[3] Jochen Liedtke. The performance of μ-kernel-based systems. In *Proceedings of SOSP '97*, 1997.

[4] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[5] Larry McVoy. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*. Usenix, 1996.

[6] Motorola and IBM. *PowerPC 603 User's Manual*. Motorola, 1994.

[7] Theodore Romer, Wayne, Ohlrich, Anna Karlin, and Brian Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[8] Ed Silha. *The PowerPC Architecture, IBM RISC System/6000 Technology, Volume II*. IBM Corp., 1993.

[9] Richard L. Sits. Alpha axp architecture. *Communications of the ACM*, February 1993.

[10] Mark Swanson, Leigh Stoller, and John Carter. Increasing tlb reach using superpages backed by shadow memory. In *Computer Architecture News*, 1998.

[11] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address space. In *Proceedings of SOSP '95*, 1995.

[12] Shreekant S. Thakkar and Alan E. Knowles. A high-performance memory management scheme. *IEEE Computer*, May 1986.